楽な方に流れることでテストしやすいコードが生まれる

テスト駆動開発で組み込みソフトウェアの品質を上げる

細谷泰夫

テスト駆動開発（TDD：test driven development）は，eXtreme Programing（XP）という開発手法で提唱されているプラクティス（実現手段）の一つです．そう聞くと，「Javaなどには適用できても，組み込みソフトウェア開発には向いていないのでは？」と思う方がいるかもしれませんが，テスト駆動開発は，組み込みソフトウェア開発にも非常に有効な開発手法です．筆者自身も通信システムにおける組み込みソフトウェア開発にテスト駆動開発を適用し，大きな効果を実感しました．ここでは，テスト駆動開発の実践方法を具体的に紹介します．（筆者）

テスト駆動開発（TDD：test driven development）は，テストを実装の中心に据えたプログラム開発手法です．具体的には，実装対象であるプログラムのコードを書く前にテスト・プログラムのコードを実装し，テスト・プログラムに合格するように実装対象プログラムのコードを実装していきます（**図1**）．テスト・プログラムを少し書いては，対応する実装対象プログラムのコードを書くという作業を繰り返し，プログラムをあるべき姿に徐々に近づけていきます．
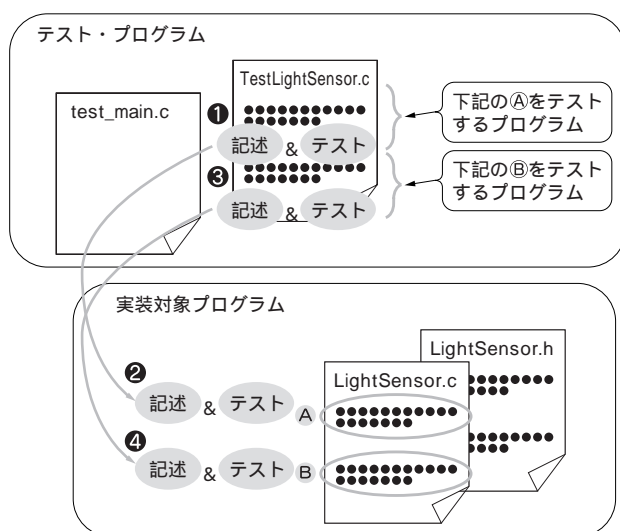
図1　テスト駆動開発の概要
まずテスト・プログラムのコードを実装し，そのテストに合格するように実装対象プログラムのコードを実装する，という作業を繰り返す．

テスト駆動開発はよくテスト手法と間違えられますが，そうではなく，あくまでも開発手法です．テスト駆動開発で行う「テスト」は，コードを実装する過程において実装者がどのように考えたのかを整理した結果であり，いわゆる「単体テスト」とは別のものです．このことを理解した上で，単体テストを効率良く実施するための方法としてテスト駆動開発を活用することは可能です．なぜなら，単体テストとテスト・コードは多くの場合，ほとんど同じ内容となることが多いからです．テスト駆動開発のテスト・コードをレビューしたり，カバレッジ・ツールを活用してテスト・コードの網羅性を保証したり，テスト・コードを補完するテストを追加することなどによって，テスト駆動開発を単体テストに代えることも可能です．

まずはテスト駆動開発を実践し，有用性を理解してみてください．そうすれば活用方法はおのずと見えてくるでしょう．

● テスト駆動開発のために必要な環境

C言語でテスト駆動開発を実践するためのツールとして

KeyWord　TDD，開発手法，テスト，CUnit for Mr.Ando，リファクタリング，オブジェクト指向，ETロボコン

は，「CUnit」や「CUnit for Mr.Ando」[1][2]というテスト・フレームワークが用意されています．CUnitのほうが有名で多機能，CUnit for Mr.Andoのほうがシンプルで導入しやすい，という特徴があります．本稿では，CUnit for Mr.Andoに多少変更を加えたものを使用します．

CUnit for Mr.Andoは，テスト用の関数testRunnerを含んだソース・ファイルとヘッダ・ファイルを備えています(**図2**)．テスト用のソース・ファイル(test_main.cやtestLightSensor.cなど)にコードを記述し，そのテストに合格するように実装ファイル(LightSensor.cなど)を記述していきます．

C言語でテスト駆動開発を実施するとき，各クラスのテストごとにスタブ(テスト用の疑似関数)が競合するという問題が起こるので，CUnit for Mr.Andoなどでは各クラス(光センサ，ライン・センサなど)の単体テストごとにコンパイル環境(Makefile)を分けることを推奨しています．筆者はスタブを外部から設定するインターフェースを実装することにより，コンパイル環境を分けずに(一つのMakefileで)テストできるようにしました．また，今回扱う例はシンプルであるため，**図2**に示したファイルを一つのフォルダにまとめることもできます(スタブの競合については，下掲のコラム「C言語によるテスト駆動開発の問題点」を参照)．

図2　CUnit for Mr.Andoのフォルダ構成例
テスト用ソース・ファイルに確認したい内容を記述する．テストそのものは，testRunner.cなどに定義されたテスト用の関数を利用して実施する．なお，本稿では，筆者がスタブを外部から設定するインターフェースを実装しており，コンパイル環境を分ける必要がないので，フォルダ構成はもっと単純にできる(例えば，すべてを一つのフォルダにまとめることも可能)．

● テスト駆動開発の基本的な流れ

まず，テスト・フレームワークを用いてテスト駆動開発を行うための基本手順について理解しましょう．ここでは，今から実装しようとしているクラスを「実装クラス」，実装クラスに対してテストを記述するクラスを「テスト・クラス」と呼びます．テスト駆動開発の基本的な流れは**図3**のようになります．

コンパイルが通らなかったり，テストが失敗したりする状態を「レッド」，テストがすべて成功している状態を「グリーン」，そしてプログラムの動作を変えずに構造を手直しすることを「リファクタリング」と呼びます．

通常のプログラミングでは，正しく動作しているコード

COLUMN - 1　C言語によるテスト駆動開発の問題点

C言語によるテスト駆動開発を可能とするテスト・フレームワークとしては，「CUnit」や「CUnit for Mr.Ando」があります．しかし，フレームワークはあるのに，C言語によるテスト駆動開発はあまり普及していません．

その理由の一つとして考えられるのは，言語として回帰テストを行うことが困難であることです．テスト駆動開発で，ある関数(クラスA)からある関数(クラスB)を呼ぶことをテストする場合，クラスAのテスト結果がクラスBの実装に影響されないように，クラスBの実体を呼び出すのではなくクラスBのダミーを用いることになります．JavaやC++などのオブジェクト指向言語では，クラスBをオーバライドすればよいのですが，C言語ではスタブ(テスト用の疑似関数)を用いてテストすることになります．このとき，スタブと実体の名称が2重定義となってしまうため，スタブと実体ではコンパイル環境を分ける必要が出てきます．

テスト駆動開発で重要なのは，「レッド―グリーン―リファクタリング」の軽快なリズムと，常に回帰テストが行えるからこそ可能なリファクタリングによるクラス設計の改善です．コンパイル環境がいくつにも分かれた状態では，これらを実現するのは難しいでしょう．

筆者が参画している日本XPユーザグループ(XPJUG)関西支部組込みTDD分科会は，C言語でテスト駆動開発を行うための具体的な手法を確立するために活動しています．スタブの競合に対しては，スタブを外部から設定するインターフェースを実装することにより，コンパイル環境を分けずに回帰テストができるようにする方法を提案しています[3]．この資料は本誌の付属CD-ROMにも収録しているので，参照してください．

はなるべく変更しないという習慣があります．しかし，テスト駆動開発では，グリーンの状態を保っていることを確認しながらリファクタリングでプログラムの資産価値を向上させることができます．この「レッド－グリーン－リファクタリング」を短い周期で行うことにより，プログラミング作業にリズム（めりはり）が生じます．テスト駆動開発では，このリズムに乗って開発を進めることが重要です．
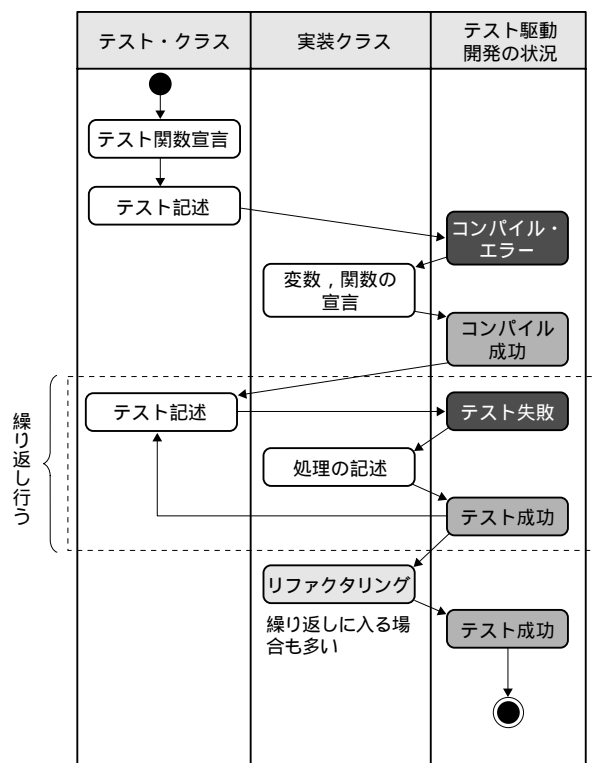
図3　テスト駆動開発の基本的な流れ

テストの記述，テスト実行，処理の記述，テスト実行，…と繰り返し，短い周期でリズミカルに「テスト失敗（レッド）」と「テスト成功（グリーン）」を経験しながらプログラミングを進めていく．

ライン・トレース・ロボットの開発

それでは，実際にテスト駆動開発による開発を体験してみましょう．開発対象は**図4**のような特徴を持つライン・トレース・ロボットとします．このライン・トレース・ロボットの制御ソフトウェアを，自律オブジェクト指向を適用してモデリングした結果が，**図5**の静的モデルです(4)．ここでは，このモデルのうち光センサによるライン認識の一部を，テスト駆動開発で実装してみます（C言語におけるクラス実装については，下掲のコラム「組み込みにおけるオブジェクト指向とクラス」を参照）．

光センサによるライン認識の動的モデルは，例えば**図6**
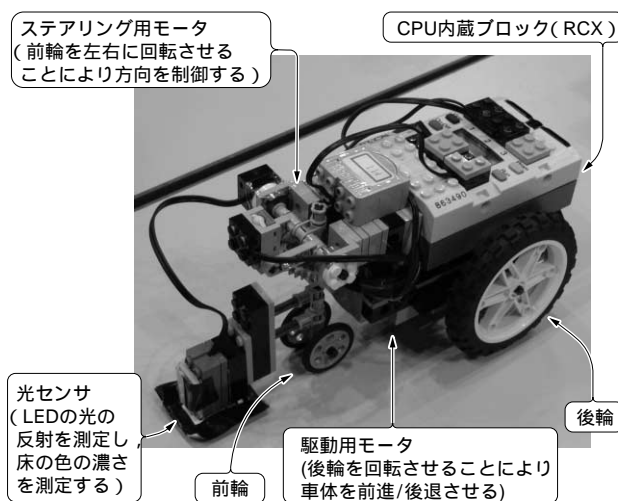
図4　開発対象であるライン・トレース・ロボット

デンマークLEGO社のロボット・キット「LEGO MINDSTORMS」で構成されるライン・トレース・ロボット．ETソフトウェア・デザイン・ロボット・コンテスト（通称ETロボコン）の走行体として規定されているものである．

COLUMN-2　組み込みにおけるオブジェクト指向とクラス

オブジェクト指向やUML（unified modeling language）においては，「データとそのデータに関連する振る舞いが一体となった概念」をクラスと呼びます．「オブジェクト指向はリソースの限られた組み込みシステム開発には向いていないのでは？」と思われるかもしれません．しかし，さまざまな装置が連携するシステム（例えば通信システムなど）において，必要なハードウェアを設計することを想像してみましょう．このとき，それぞれの装置に必要なデータと機能をセットにして役割分担を決定します．この「装置」にあたるのが「クラス」であると言えます(4)．ソフトウェア設計におけるオブジェクト指向は，ソフトウェア上で仮想的な概念（装置など）を設計し，それぞれの役割分担にふさわしいデータと振る舞いを割り当てることだと言えます．

ところで，オブジェクト指向言語ではないC言語で，どのようにクラスを実装すればよいでしょうか． 実は，私たちはC言語においても無意識に「クラス」をいつも扱っています．C言語の`FILE`ポインタとファイル操作関数を考えてみましょう．私たちは`FILE`ポインタのメンバに直接アクセスせず，ファイル操作用の関数を介してメンバを操作します．この`FILE`ポインタとファイル関数のセットは，まさに上記のクラスの定義と合致します．

本稿では，構造体を定義し，ポインタを用いてそれを生成したりメンバ変数にアクセスしたりすることで，C言語におけるクラス実装を実現しています．

のように分析できます．ここでは，これらのモデルに基づいて実際に光センサ・クラスを実装してみましょう．

● 環境構築

まず，テスト駆動開発の環境を整えます．本誌の付属CD-ROMに収録されているデータの中から，本稿記事関連データのdata\srcフォルダ内のファイルをお手元のCプログラム開発環境にコピーし，プロジェクト（統合開発環境でない場合はMakefile）に追加してください．必要な準備はそれだけです．

なお，開発環境がなくても，本稿をざっと読めばテスト駆動開発のおおよその感覚をつかむことはできると思います．

● テスト・クラスを追加する

まず，光センサ・クラスに対応するテスト・クラスを新たに追加します．そのためには，テスト・プロジェクトのmain関数を実装しているソース・ファイル（本稿で言うとtest_main.c）にテスト関数の記述を追加するところから始めます（**リスト1**）．

ここで一度ビルドを実行してみます．「undefined reference to `_Runner_TestLightSensor'」と表示され，コンパイル・エラーが発生します．まだテスト・クラス（関数Runner_TestLightSensorの内容）を定義していないので，エラーが発生するのは当然です．これは，「レッド－グリーン－リファクタリング」のレッドに当たる部分です．

とりあえずの目標は明確です．コンパイル・エラーを解消し，グリーンとなる方法を考えます．そのためには，テスト・プロジェクトのソース・ファイルに**リスト2**のような関数を宣言すればよいと考えられます．

一度実行してみましょう．「Total Result OK:0 NG:0」と表示されます．まだテストを書いていないのでOKもNGも0個ですが，とりあえずエラーは解消し，グリーンの状態になりました．

ここまでがテスト・クラスを追加する手順となります．
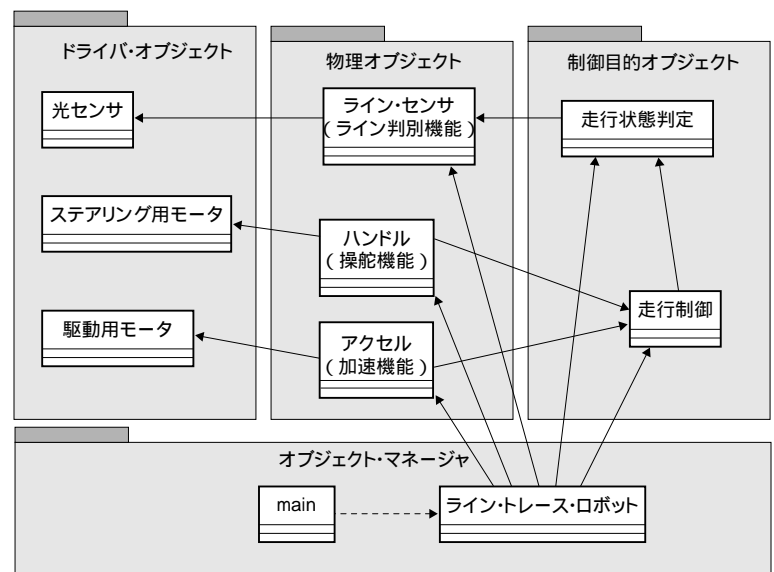
図5
ライン・トレース・ロボットのクラス図（静的モデル）
多重度はすべて1対1，ロール名はクラス名と同じとする．
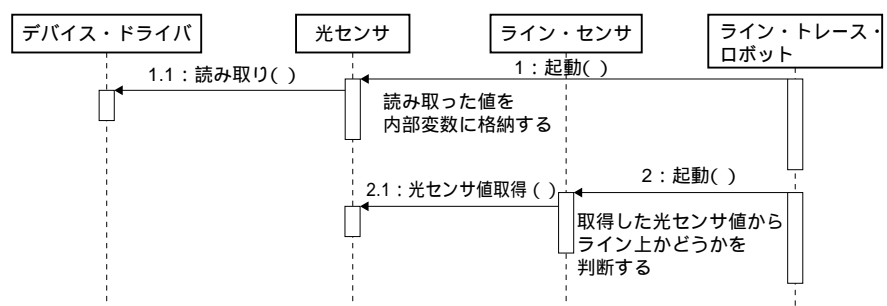
図6
ライン・トレース・ロボットのシーケンス図（動的モデル）
静的モデルのドキュメントに基づいて実装する場合も，スケッチ程度でもよいので動的モデルの分析を行うべきである．このような場合，動的モデルは実装中に変更する可能性があるので，手書きのスケッチで十分である．グループで開発する場合は動的モデルについて議論し，その結果描き上がったホワイト・ボードの図をコピーするとよいだろう．

以降は，「テスト記述→実装」を繰り返すことによって，必要な機能を実装クラスに作り込んでいきます．

● コンストラクタとデストラクタを記述する

それでは，光センサ・クラスを実装してみましょう．

1）コンストラクタの実装

テスト・クラスに定義した関数Runner_TestLightSensorに，光センサ・クラスのコンストラクタ注1に対するテスト関数を記述します（**リスト3**）．ここで登場するtestRunnerという関数は，CUnit for Mr.Andoにおけるテスト関数実行のための関数です．testRunner関数の第1引き数は実行するテスト関数の関数ポインタで，第2引き数はテスト関数の名称を表す文字列です．

とりあえずコンパイルすると，「undefined reference to `_CTestLightSensor_Constructor'」と表示されました．コンストラクタが定義されていないのでコンパイル・エラーになります．またレッドの状態になりました．

そこで，コンストラクタのテスト関数を宣言します（**リスト4**）．これでコンパイルが通るようになりました．再びグリーンです．

このように，リファクタリングの必要がない場面では，レッド→グリーンを繰り返します．この繰り返しが，テスト駆動開発独特のリズムを生み出します．

注1：コンストラクタとは，オブジェクト指向言語において，オブジェクト（クラスの実体など）を生成するための特殊な関数である．クラスの実体を生成するときは，必ず最初にこの関数が呼び出される．また，対になる関数として，クラスの実体を消滅するときに呼び出される関数デストラクタがある．

リスト1　新たなテスト・クラスのプロトタイプを定義する（test_main.c）

```
#include "testRunner.h"

#define COUNT_TESTModule    1 //テスト・クラスの数

/*ここにモジュールごとのテスト実行関数のプロトタイプを定義する*/
void Runner_TestLightSensor(void);
typedef void (*RNNER_TEST_FUNC)(void);


/*ここにモジュールごとのテスト関数を登録する*/
RNNER_TEST_FUNC listRunnerTest[]=
{
    Runner_TestLightSensor
};


int main(int argc, char* argv[])
{
    /*モジュールごとのテストを順番に実行する*/
    long i;
    InitCounter();
    for (i=0;i<COUNT_TESTModule;i++)
        listRunnerTest[i]();

    DisplayCounter();
    return 0;
}
```


リスト2　定義したテスト実行関数を宣言する（TestLightSensor.c）

```
#include "testRunner.h"

void Runner_TestLightSensor()
{

}
```

リスト4　記述したコンストラクタのテスト関数を宣言する（TestLightSensor.c）

```
unsigned int  CTestLightSensor_Constructor(void)
{

}
```

リスト3　実装クラスのコンストラクタに対するテスト関数を記述する（TestLightSensor.c）

```
void Runner_TestLightSensor()
{
    testRunner(CTestLightSensor_Constructor,"CTestLightSensor_Constructor");
}
```

いよいよコンストラクタをテストするコードを実装します．コンストラクタに期待する動作とはどのようなものかを考えましょう．でも，一度にすべてに取り組む必要はありません．少しずつ考えていきましょう．

まず，コンストラクタというからには，オブジェクトを生成するはずです．これをテストで表現すると，**リスト5**のようになります．ここで，CUnit for Mr.Andoで提供されているTEST_ASSERTというマクロを用いて，テスト結果を判定しています．TEST_ASSERTマクロは，パラメータとして与えられた論理式が真の場合は「テスト成功」，偽の場合は「テスト失敗」という判定を行います．つまり，**リスト5**のテスト・コードは「コンストラクタによってCLightSensorのインスタンス（クラスの実体）が返された結果，ポインタにNULLでない値が格納されていること」を確認するものです．

この状態でコンパイルを行うと，CLightSensor，CLightSensor_Constructorが宣言されていないのでコンパイル・エラーが出ます．クラスを定義してコンパイル・エラーを解決しましょう（**リスト6**）．なお，クラスの型は構造体として宣言します．

このヘッダ・ファイルをテスト・クラスにインクルードすればコンパイルが成功するので，テストを実行しましょう．結果は「../TestLightSensor.c:8: error: TestCaseError（0x00000000） CTestLightSensor_Constructor NG （now 0 ok...） Total Result OK:0 NG:1」となり，テストが失敗します．このように，テストを書いてコンパイルが通った後に，テストを実行して失敗するのを確認しましょう．わざとレッドの状態を確認することによって，テストが実行されていることを確認できるし，これから解決するべき課題を明確にできます．

では，上記エラーを解決し，グリーンの状態にしましょう．コンストラクタにオブジェクト生成処理を追加します（**リスト7**）．早速テストを実行しましょう．「CTestLightSensor_Constructor OK （ 1 tests） Total Result OK:1 NG:0」と表示されます．これで，最初のテストに成功しました．

2）デストラクタの実装

デストラクタについてもコンストラクタと同様に実装しましょう．デストラクタに関するテストにおいては，コンストラクタで生成されたポインタがデストラクタで解放され，NULLが格納されていることを確認します．

実装するテスト・コードを**リスト8**に示します．実際は，

リスト5　コンストラクタのテスト関数を実装する（TestLightSensor.c）

```
unsigned int  CTestLightSensor_Constructor(void)
{
    CLightSensor* p=NULL;
    p=CLightSensor_Constructor();
    TEST_ASSERT(NULL!=p);
}
```

リスト6　CLightSensorクラスとコンストラクタを宣言する

```
typedef struct
{

}CLightSensor;
```

（a）クラスを構造体として宣言する（LightSensor.h）

```
CLightSensor* CLightSensor_Constructor(void)
{
    return NULL;
}
```

（b）クラスのコンストラクタを宣言する（LightSensor.c）

リスト7　コンストラクタにオブジェクト生成処理を実装する（LightSensor.c）

```
CLightSensor* CLightSensor_Constructor(void)
{
    CLightSensor* p=
       (CLightSensor*)malloc(sizeof(CLightSensor));
    return p;
}
```

リスト8　デストラクタのテスト関数を記述，宣言，実装する（リスト3～リスト5に対応）

```
void Runner_TestLightSensor()
{
    testRunner(CTestLightSensor_Constructor,
               "CTestLightSensor_Constructor");

    testRunner(CTestLightSensor_Destructor,
               "CTestLightSensor_Destructor");
}
```

（a）実装クラスのデストラクタに対するテスト関数記述を追加する（TestLightSensor.c）

```
unsigned int  CTestLightSensor_Destructor(void)
{
    CLightSensor* p=NULL;
    p=CLightSensor_Constructor();
    TEST_ASSERT(NULL != p);

    CLightSensor_Destructor(&p);
    TEST_ASSERT(NULL == p);

    return 0;
}
```

（b）デストラクタのテスト関数を実装する（TestLightSensor.c）

コンストラクタの時と同じように，コンパイル・エラー（レッド）→修正（グリーン）を繰り返しましょう．

テストを実行すると，デストラクタを実装していないためコンパイル・エラーとなります．そこで，次に実装コードのほうを実装します．一気にテストが成功する実装を書いてしまいたい衝動に駆られるかもしれませんが，一歩ずつ進んでいきましょう．一見効率が悪いように思えますが，テスト駆動開発のリズムを身に付けるには，少しずつ考えながら進んでいく習慣を付けるのが早道です．デストラクタの宣言のみ行って（**リスト9**），テストを実行します．「../TestLightSensor.c:19: error: TestCase Error (0x00000000) CTestLightSensor_Destructor NG (now 1 ok...) Total Result OK:1 NG:1」と表示されます．デストラクタの処理を書いていないので，まずはテストが失敗します．それからデストラクタの処理を書いて（**リスト10**），再度テストを実行してみましょう．「CTestLightSensor_Constructor OK (1 tests) CTestLightSensor_Destructor OK (2 tests) Total Result OK:2 NG:0」と表示されます．テスト成功です．

これでコンストラクタとデストラクタの実装が完了しました．ここまでの進め方は，すべてのクラスについて共通です．

● 光センサの測定値取得部分を実装する

次に，光センサの測定値取得部分を実装してみましょう．まず，光センサ・クラスの動作の概略を以下のように定義します．

- 関数CLightSensor_Activateが呼び出されたら，センサ値をデバイス・ドライバより取得し，内部変数に設定する
- 関数CLightValue_GetLightValueが呼び出されたら内部変数に格納されている値を返す
- 未測定状態でのセンサ値は0とする

1）テスト用にデバイス・ドライバのラッパを用意する

実機に依存しないテストを行うために，デバイス・ドライバのラッパを用意します（**リスト11**）．#ifdefを用いて，テスト時にはデバイス・ドライバを参照せず，デフォルトの値を返すようにします．

2）CLightSensor_Activate関数の実装

まずは，CLightSensor_Activateに対するテスト関数を追加します．追加の方法はコンストラクタやデストラクタ（**リスト3**，**リスト8(a)**）と同様です．次に，テストを記述します．「CLightSensor_Activateが実行されれば測定値が内部変数に設定されるはず」をテストで記述しました（**リスト12**）．

なお，ここではp->light_valueという内部変数をテスト・クラスから直接参照しています．一般的に，内部変数を外部から参照することは禁止されていますが，ここではテスト・クラスだけは例外的に参照してもよいという規約を設けています．これは組み込み開発では内部的な状態の変化が重要になるため，テスト・クラスからは内部変数を参照できた方がメリットがあるという考えによるものです．

リスト9　クラスのデストラクタを宣言する（LightSensor.c．リスト6(b)に対応）

```
void CLightSensor_Destructor(CLightSensor** p)
{

}
```

リスト10　デストラクタにオブジェクト消滅処理を実装する（LightSensor.c．リスト7に対応）

```
void CLightSensor_Destructor(CLightSensor** p)
{
    free(*p);
    *p=NULL;
}
```

リスト11　テスト用にデバイス・ドライバをラッピングする(syscall.c)

```
int getLightValue()
{
#ifdef TESTPROJ
    return DEFAULT_LIGHT_VALUE;
#else
    ［デバイス・ドライバからの値の取得］
    return ［デバイス・ドライバから取得した値］
#endif
}
```

リスト12　Activateのテスト関数を実装する（TestLightSensor.c）

```
unsigned int  CTestLightSensor_Activate(void)
{
    CLightSensor* p=NULL;
    p=CLightSensor_Constructor();
    TEST_ASSERT(NULL != p);

    CLightSensor_Activate(p);
    TEST_ASSERT(DEFAULT_LIGHT_VALUE==
    p->light_value);

    CLightSensor_Destructor(&p);
    TEST_ASSERT(NULL == p);
    return 0;
}
```

ここでコンパイルすると，CLightSensor_Activateやp->light_valueが定義されていないというエラーが発生するので，実装します．エラーに引っ張られて実装するという感覚が出てくれば，テスト駆動開発のリズムを体感できていると言えるでしょう．

とりあえず，コンパイルを通す宣言を行いテストを実行すると「CTestLightSensor_Activate NG (now 1 ok...)」と表示され，テストが失敗します．テストを成功させるには，CTestLightSensor_Activateにおいて，センサ値を設定する必要があります．

先ほど#ifdefでラッピングしたデバイス・ドライバのセンサ値取得関数getLightValueにより，センサ値を取得し，内部変数に代入しましょう（**リスト13**）．テストを実装するとNGが消えているはずです．

3）CLightSensor_GetLightValue関数の実装

光センサの値を取得する関数（CLightSensor_GetLightValue）が内部変数に格納されているセンサ値を返すことをテストします．CLightSensor_Activateのテスト関数（**リスト12**）を参考に，**リスト14**を記述します．関数CLightSensor_GetLightValueの実装は**リスト15**のように，内部変数を返却します．

最後の仕上げとして，測定前は0を返す必要があるため，コンストラクタにメンバ変数初期化処理を実装します（**リスト16**）．これですべてのテストが成功します．

ここまでに，一つのクラスをテスト駆動開発で実装できました．このクラスで実践したテスト駆動開発の基本的な流れや，コンストラクタ，デストラクタのテスト方法は，ほかのクラスを実装する場合も同様です．ぜひ，日常のプログラミングで試してみてください．テスト駆動開発の楽しさ，効果を実感できるでしょう．

テスト駆動開発の効果

テスト駆動開発を実践してどのように感じたでしょうか？ 慣れないうちは，テストを書く分だけ効率が悪いように感じるかもしれません．しかし，筆者の体験によると，テスト駆動開発には下記のような効果があります．

● 効果その1：コーディングが楽しくなる

筆者は関西人なので，書いたコードがちゃんと動いているかをすぐに知りたいと思います（要するに，せっかちなのだ）．テスト駆動開発では「テスト失敗―実装―テスト成功」を短いスパンで繰り返すため，書いたコードの結果がすぐに分かります．これが一種のゲームのような感覚とな

リスト13　CLightSensor_Activate関数を実装する（LightSensor.c）

```
void CLightSensor_Activate(CLightSensor* p)
{
    p->light_value=getLightValue();
}
```

リスト14　GetLightValueのテスト関数を実装する（TestLightSensor.c）

```
unsigned int  CLightSensor_GetLightValue(void)
{
    CLightSensor* p=NULL;
    p=CLightSensor_Constructor();
    TEST_ASSERT(NULL != p);

    /*測定前は0を返す*/
    TEST_ASSERT(0==CLightSensor_GetLightValue());

    /*測定後はデフォルト値を返す*/
    CLightSensor_Activate(p);
    TEST_ASSERT(DEFAULT_LIGHT_VALUE
    ==CLightSensor_GetLightValue());

    CLightSensor_Destructor(&p);
    TEST_ASSERT(NULL == p);
    return 0;
}
```

リスト15　CLightSensor_GetLightValue関数を実装する（LightSensor.c）

```
int CLightSensor_GetLightValue(CLightSensor* p)
{
    return p->light_value;
}
```

リスト16　コンストラクタにメンバ変数初期化処理を実装する（LightSensor.c）

```
CLightSensor* CLightSensor_Constructor(void)
{
    CLightSensor* p=(CLightSensor*)malloc(sizeof(CLightSensor));
    if (p!=NULL)
        p->light_value=0;

    return p;
}
```

COLUMN-3 XPと組み込みシステム開発

テスト駆動開発は，アジャイル開発手法であるeXtreme Programing（XP）のプラクティス（実現手段）の一つとして提唱されています．XPは「仕様変更に素早く対応する必要がある」，「最終的にどうなればよいかが分からない」というような開発に向いています．反面，品質保証や請負金額の固定化が重視されるような開発には不向きと言えます．

筆者はIT系のソフトウェア開発と組み込みソフトウェア開発を両方経験していますが，組み込み開発においてはシミュレータなどのテスト用プログラムを作成する機会が多いと感じています．また，ハードウェア設計者も，意外とテスト用のソフトウェアを開発している人が多いです．筆者の経験上，このようなテスト・プログラムの作成にXPなどのアジャイル開発の要素を取り入れることは有効だと感じています．

また，XPはコミュニケーションやモチベーションといったいわゆる人間系を重視するプロセスであり，プラクティスにはそれらを高めるような工夫がされています．筆者は，ソフトウェアの開発プロセスとは別に，マネジメントの手法としてXPを活用できないかと考えています．開発プロセス自体はウォータ・フォール・モデルでも，素早い意思決定や関係者（ステーク・ホルダ）同士の連携，手軽に状況を「見える化」する手段として，XPを取り入れることができると考えています．

り，コーディングが楽しくなります．

● 効果その２：デバッグの時間が減る

細かいスパンで動作を確かめながら開発を進めていくので，論理的な誤りによるバグが混入しにくくなります．また，絶えずテストを実行しているため，問題が発生した場合も前回のテスト以降に実装したコードを確認すればよく，原因の特定が短時間で済みます．筆者はテスト駆動開発を適用してから，デバッガを用いる回数が1/10程度になりました．

● 効果その３：プログラムの体質が改善する

テスト駆動開発を適用すると，自然と「テストしやすいコード」を記述するようになります．人間はどうしても楽な方に流れる傾向がありますが，テスト駆動開発の枠内で楽をするには，グローバル変数や不必要に密な関連などテストを書きにくいコードは敬遠するようになり，テストしやすい，結合度が疎なクラスの集合体が出来上がります．

筆者が静的解析を行った経験によると，静的経路数や関数あたりの行数はかなり減ります．

● 業務にテスト駆動開発を適用するための上司説得術

テスト駆動開発は適用の仕方に柔軟性があり，個人的な実装改善方法としても十分に効果がある手法です．しかし，プロジェクトとして組織的にテスト駆動開発を適用したい場合には，上司を説得する必要があるかもしれません．eXtreme Programing（XP）やテスト駆動開発を知らない上司を説得するには，彼らの言葉で有用性を説明してあげる必要があります．

以前どこかで，「XP導入を上司に説得するには，PDCA（plan，do，check，action）サイクルをしっかりと回したい，と言えばよい」と聞いたことがあります．テスト駆動開発はまさに「テスト記述（plan）」，「実装（do）」，「テスト実行（check）」，「コード修正（action）」というPDCAサイクルをコード実装に取り入れた手法だと言えます．「PDCAサイクルを回すことで，実装の品質を効率的に向上させる手法です」と説明してみてはいかがでしょうか．

参考・引用*文献

(1) CUnit for Mr.AndoのWebページ（概要），http://park.ruru.ne.jp/ando/work/CUnitForAndo/html/index_ja.html
(2) CUnit for Mr.Andoのダウンロード・ページ，http://sourceforge.jp/projects/cunitforando/
(3) XPJUG関西支部 組込みTDD分科会；組込みでもTDD！，XP祭り関西2006，2006年9月．
(4) 岩橋正実；リアルタイムシステム実現のための自律オブジェクト指向，CQ出版社，2002年4月．

ほそたに・やすお
日本XPユーザグループ関西支部　組込みTDD分科会

<筆者プロフィール>

細谷泰夫．2001年，2度目の転職により現在の某電機メーカに入社．以来，基幹系通信システムのソフトウェア設計に従事．個人的な活動としてXPJUG関西支部運営委員として組込みTDD分科会を主催，C言語におけるテスト駆動開発実践方法確立に向けて活動している．最近は開発における人間系の重要さを痛感し，プロジェクト・ファシリテーションに興味を持っている．